Project 1: Quadratic Sieve in Rust

Group 06: Fredrick Nilsson

November 24, 2023

Exercise 1

Assume that we can perform ten million tests of the above kind each second on our computer. How long would it take to factor a 25 digit number (with two prime factors both of the same order (12 digit numbers))?

If we assume that the 25-digit number is the largest posssible, 9,999,999,999,999,999,999,999,999, and that both of the two prime factors are the smallest possible 12-digit prime number, 100,000,000,003.

$$9999999999999999999999999/(100000000003^2) \approx 1000$$

Then the remaining prime factors have has to be smaller than 1000. So we can assume that these factors will be discovered in the first 0.0001 seconds of the program.
This only leaves the two 12-digit primes to be discovered. Which will take quite a bit more time.
But when the first 12-digit prime factor is found, we can easily figure out the last factor by dividing the 25-digit number by the known factors, therefore we only need to find one of them.
As the smallest 12-digit prime number is 100 000 000 003 it will in the best case take

$$100000000003/10000000 = 1000.00000003$$

seconds for it get solved. Since the largest 12-digit prime number is 999 999 999 961 it will in the worst case take

$$999999999961/10000000 = 9999.99999961$$

seconds for it get solved.
If we assume that the rest of the 12-digit primes have a close to even distribution between the best and worst case, we can assume that the average time it takes to find a 12-digit prime factor is

$$(1000.00000003 + 9999.99999961)/2 = 5499.99999982$$

seconds.

Exercise 2

If you want to factor many 25 digit numbers, you can improve the running time of the basic trial division by first precomputing and storing the primes up to

$$\sqrt{N}$$

How much faster does your improved trial division algorithm become for 25 digit numbers? Roughly how much storage does your algorithm require? What kind of budget does the storage requirement demand; student budget, big government grant, more dollars than there are atoms in the universe? You may check current storage pricing on http://www.prisjakt.nu.

If we assume that the 25-digit number is the largest posssible, 9,999,999,999,999,999,999,999,999, then the square root of that number is approximately 3,162,277,660,168. So we need to precompute and store all the primes up to 3,162,277,660,168.
According to the prime number theory[1], the amount of primes smaller than 3,162,277,660,168 can be approximated as

$$\frac{3,162,277,660,168}{ln(3,162,277,660,168)} \approx 109,868,779,044$$

.
As the largest prime that could possibly be stored is smaller than 3,162,277,660,168, we can assume that no number prime will need more than 42 bits to store, since 3,162,277,660,168 = 101110000001000110011011111100011000001000 in binary. If we then assume that all the primes will be stored in 42 bits each, the total size to store all the primes is

$$109,868,779,044 * 42 = 4,614,488,719,848$$

bits. Which is approximately 577 gigabytes. Although this is assuming a close to optimal way to store the primes, so we can assume that the actual storage size will be a bit larger.
According to prisjakt.nu, we can buy an 1TB Seagate Barracuda ST1000DM010 harddrive for 559 sek. So we can assume that the storage requirement can be satisfied by a student budget.

Exercise 3

Factoring my number, 127423368713324519534591, I found the factors 312709043917 and 407482198523.

It took 14458 ms (14s 458ms) for the program to find them on a Macbook Air (2020).

In order to find the factors, I employed an implementation of the quadratic sieve algorithm[2] in Rust. The algorithm is largly based on the description found in the project description.

It begins by running the *exercise3* (line 5) function, where inside the function you have to pre-set the number you want to factor on line 12. The algorithm itself being with the calling of the function *find_factors* (line 23), which simply find 2 factors of the input number refered to as N from now on. It does this by first deciding on a bound B, that is set as double the root of N, rounded into the span of of 10000-50, as any numbers larger than that will use too much memory, and any smaller might break the program for a small N. Then it generates all primes lower than the square root of N with the function *gen_primes*(line 150), which is a simple implementation of the Sieve of Eratosthenes[3]. These primes are then filtered so that we only keep primes that are quadratic residues of N through the function *legendre*(line 174). The filtering is done in order the decrease the amount of primes we are working with in order too speed up the algorithm and reduce the memory usage. This should not cause a problem when it comes to generation of B-smooth numbers, as we are only interested in the quadratic residues of N.[2]
Once we have our primes the main time-sink of the algorithm is the generation of B-smooth numbers, which starts at line 45. In order too speed up this process, I employ multithreading, letting 8 different threads find these numbers at the same time. Originally I employed the formula

$$r = \lfloor \sqrt{N * k} \rfloor + j$$

and

$$r^2 \mod N$$

described in the project description to find and check B-smooth numbers, but I found that this was too inefficient and slow. So I instead used the formula

$$r = \lfloor \sqrt{N} \rfloor + j$$

which allowed me to only calculate the square root once, it also allowed me to get rid of the modulo operation, as

$$r^2 = N + 2 * j * \sqrt{N} + j^2$$

which makes it so that when

$$0 \leq j < (\sqrt{2} - 1)\sqrt{N} \implies r^2 \mod N = 2 * j * \sqrt{N} + j^2$$

So for a decently large N, we can replace the modulo operation with a multiplication and a subtraction. These changes approximately reduces the run-time of the algorith to

one fifth of the original, although it makes the algorithm fail for some small numbers. These calculations are performed in the different threads at line 54-67.
All of the generate numbers are checked if they are smooth to our prime-base by the function *check_smoothness* (line 125).

When we have all out B-smooth numbers, we can start figuring out if any combinations of them can have even exponants on all their prime factors by constructiong a matrix of their prime factors exponants modulo 2 at line 75-80. We find the possible solutions by gauss reduction, done by the function *find_solutions* (line 183), which formats the data and calls upon a recursive function *recursive_gauss* (line 200) to do the actual reduction.
The *recursive_gauss* function works by first finding the first non-zero row, and then subtracting that row from all the other rows that have a 1 in the same column. It then calls itself recursively with the reduced matrix, and the next column. When the left hand matrix is reduced to purely zeroes, we assume that the rows matching on the right-hand matrix that match are possible solutions, and therefore return these.

These possible solutions consists of both the B-smooth numbers and the prime factor exponants of their quadratic modulo form. Using these, we calculate the numbers a and b at line 82-102. Due to them being a solution we know the formula

$$a^2 = b^2 \mod N$$

is true. Therefore we can calculate some factor of N by employing

$$\gcd(|a - b|, N)$$

This will give some factor of N, although it can be the factor 1 or N itself. Therefore we employ a check that the factor are neither of these numbers, if it is we instead try the next solution. If we find a factor that is not 1 or N, we can assume that we have found a factor of N, and therefore we can calculate the other factor by dividing N by the found factor. Returning both of these factors the *exercise3* function, finishing the program.

Exercise 4

Factoring the number 92434447339770015548544881401 resulted in the factors 727123456789451 and 127123456789451.

It took 388201 ms (6m 28s 201ms) for the program to find them on a M1 Macbook Air (2021).

For exercise 4 the same algorithm as in exercise 3 was used, with no changes.

Program printout for exercise 3

```
[2023-11-24T13:49:51.324Z INFO  project1::exercise4] Let's find some factors of N=1274233687133245195345 91
[2023-11-24T13:49:51.324Z INFO  project1::exercise4] Got bound B = 10000
```

```
[2023-11-24T13:49:51.421Z INFO  project1::exercise4] Generating B-smooth numbers, please wait
[2023-11-24T13:50:05.780Z INFO  project1::exercise4] Found some factors: 312709043917, 407482198523 => n =
    312709043917*407482198523 = 127423368713324519534591
[2023-11-24T13:50:05.781Z INFO  project1::exercise4] Two factors of 127423368713324519534591 are
    312709043917 and 407482198523
Took 14458ms
```

Working hours

I used approximately 1 hour for exercise 1 and 2. But exercise 3 and 4 took approximately 25 hours as I had to rewrite the program several times. Then I spent approximately 2 hours writing this report. Total time spent: circa 28 hours.

Source code

```rust
use num::{Integer, One, Zero, ToPrimitive};
use num_bigint::BigUint;
use std::{cmp, sync::atomic};

pub fn exercise3() {
    let now = std::time::Instant::now();

    env_logger::Builder::from_env(env_logger::Env::default().default_filter_or("info"))
        .format_timestamp_millis()
        .init();

    let num: BigUint = BigUint::parse_bytes(b"127423368713324519534591", 10).unwrap();

    if let Some((f1, f2)) = find_factors(num.clone()) {
        log::info!("Two␣factors␣of␣{num}␣are␣{f1}␣and␣{f2}", num = num, f1 = f1, f2 = f2);
    } else {
        log::error!("No␣factors␣found␣for␣{num}", num = num);
    }

    println!("Took␣{}ms", now.elapsed().as_millis());
}

fn find_factors(n: BigUint) -> Option<(BigUint, BigUint)> {
    // Increase b if needed
    let b = cmp::min(10000, cmp::max(2*n.to_f64().unwrap().sqrt().floor() as usize, 50));
    log::info!("Let's␣find␣some␣factors␣of␣N={n}");
    log::info!("Got␣bound␣B␣=␣{b}");
    // Find all prime numbers up to B
    let primes: Vec<BigUint> = gen_primes(b)
```

```
30          .into_iter()
31      // Filter all primes that aren't quadratic residue of n, makes
        the program a lot faster in general but makes it break for some
        small numbers.
32          .filter(|p| legendre(&n, &BigUint::from(*p)) == 1)
33          .map(BigUint::from)
34          .collect();
35      log::debug!("Filtered prime list is {primes:?}");
36      log::debug!("Found {p_len} primes", p_len = primes.len());
37
38      // Find enough B-smooth numbers
39      let r_vec = std::sync::Arc::new(std::sync::Mutex::new(Vec::new())
        );
40
41      let tcount = 8;
42      let prog: atomic::AtomicUsize = Default::default();
43      let max = primes.len() + 1;
44      log::info!("Generating B-smooth numbers, please wait");
45      std::thread::scope(|t| {
46          for i in 0..tcount {
47              let mut j = 1 + i as u32;
48              let prog = &prog;
49              let n = n.clone();
50              let s = n.sqrt();
51              let p = &primes;
52              let x = r_vec.clone();
53              t.spawn(move || {
54                  while prog.load(atomic::Ordering::SeqCst) < max  {
55                      // r = sqrt(N) + j
56                      // r^2 = N + 2*j*sqrt(N) + j^2
57                      // r^2 mod N = 2*j*sqrt(N) + j^2 (for small j)
58                      // works for j < sqrt(2N) - sqrt(N) = (sqrt(2) -
        1)sqrt(N) = 0.414*sqrt(N)
59
60                      let r = &s + j;
61                      let r2: BigUint = r.pow(2) - n.clone();
62                      if let Some(r2_vec) = check_smoothness(r2.clone()
        , p) {
63                          x.lock().unwrap().push((r, r2_vec));
64                          prog.fetch_add(1, atomic::Ordering::SeqCst);
65                      }
66
67                      j += tcount;
68                  }
69              });
70          }
71      });
```

```
    let r_vec = r_vec.lock().unwrap().clone();
    log::debug!("Found␣B-smooth␣numbers:␣{:?}", r_vec);

    let flat: Vec<i8> = r_vec
        .iter()
        .flat_map(|x| x.1.clone())
        .map(|x| (x % 2) as i8)
        .collect();
    let matrix = ndarray::Array2::from_shape_vec((r_vec.len(), primes.len()), flat).unwrap();
    for solution_vec in find_solutions(matrix) {
        log::debug!("Trying␣solutions:␣{:?}", solution_vec);
        let a = r_vec
            .iter()
            .zip(solution_vec.iter())
            .filter(|(_, solution)| **solution == 1)
            .map(|((r, _), _)| r.clone())
            .product::<BigUint>();
        let b = r_vec
            .iter()
            .zip(solution_vec.iter())
            .map(|((_, p_vec), solution)| {
                p_vec.iter().map(|x| *solution as u32 * (*x) as u32).collect::<Vec<u32>>()
            })
            .reduce(|x: Vec<u32>, y: Vec<u32>| {
                x.into_iter().zip(y).map(|(a, b)| a + b).collect()
            })
            .unwrap()
            .iter()
            .zip(primes.iter())
            .map(|(c, p)| p.pow(c/2))
            .product::<BigUint>();

        // a^2 - b^2 mod n = 0
        let diff: BigUint = if a > b {
            &a - &b
        } else {
            &b - &a
        };

        let f1 = diff.gcd(&n);
        let f2 = &n / f1.clone();

        log::info!("Found␣some␣factors:␣{f1},␣{f2}␣=>␣n␣=␣{f1}*{f2}␣=␣{n}");
        if f1 > One::one() && f1 < n {
```

```
116             return Some((f1, f2));
117         } else {
118             log::error!("Those␣factors␣are␣irrelevant,␣trying␣next␣
    solution");
119         }
120     }
121     return None
122 }
123 
124 // Checks if B-smooth, returns Some<Vec> of exponants if so, None
    otherwise
125 fn check_smoothness(n: BigUint, p: &Vec<BigUint>) -> Option<Vec<u8>>
    {
126     let mut r = vec![0_u8; p.len()];
127     let mut n = n.clone();
128 
129     for (i, p) in p.iter().enumerate() {
130         let mut j = 0;
131         loop {
132             let (d, r) = n.div_rem(p);
133             if r == BigUint::zero() {
134                 n = d;
135             } else {
136                 break;
137             }
138             j += 1;
139         }
140         r[i] = j;
141     }
142     if n == BigUint::one() {
143         return Some(r)
144     } else {
145         return None
146     }
147 }
148 
149 
150 fn gen_primes(n: usize) -> Vec<u32> {
151 
152     let mut primes: Vec<u32> = Vec::new();
153     let mut sieve = vec![true; n as usize];
154     let mut i = 2;
155     while i * i <= n {
156         if sieve[i as usize] {
157             let mut j = i * i;
158             while j < n {
159                 sieve[j as usize] = false;
```

```
                j += i;
            }
        }
        i += 1;
    }
    for i in 2..n {
        if sieve[i as usize] {
            primes.push(i as u32);
        }
    }
    return primes;
}

// Check if quadratic residue
fn legendre(n: &BigUint, p: &BigUint) -> u32 {
    if *p == 2u8.into() {
        return 1;
    }
    let (f, r) = (p - 1u8).div_rem(&BigUint::from(2u8));
    assert!(r == Zero::zero());
    return n.modpow(&f, p).to_u32().unwrap()
}

fn find_solutions(mut m: ndarray::Array2<i8>) -> Vec<Vec<i8>> {

    let mut right = ndarray::Array2::eye(m.nrows());

    recursive_gauss(m.view_mut(), right.view_mut());

    log::debug!("Gaussian␣reduction␣finished\n␣{:?}", m);
    log::debug!("Gaussian␣reduction␣finished␣\n␣{:?}", right);

    return m
        .axis_iter(ndarray::Axis(0))
        .zip(right.axis_iter(ndarray::Axis(0)))
        .filter(|(l, _)| l.sum() == 0)
        .map(|(_, r)| r.to_vec())
        .collect::<Vec<Vec<i8>>>();
}

fn recursive_gauss(mut left: ndarray::ArrayViewMut2<i8>, mut right:
    ndarray::ArrayViewMut2<i8>) {
    if left.is_empty() {
        return;
    }

    log::debug!("Gaussian␣reduction␣in␣progress\n␣{:?}", left);
```

```
206     log::debug!("Gaussian␣reduction␣in␣progress␣\n␣{:?}", right);
207     log::debug!("Gaussian␣reduction␣on␣matrix␣of␣size␣{:?}", left.
    shape());
208 
209     let pos = left.axis_iter_mut(ndarray::Axis(0)).enumerate().find
    (|(_ ,row)| row[0] == 1);
210 
211     if let Some((pos,_)) = pos  {
212         let mut l_col_it = left.axis_iter_mut(ndarray::Axis(0));
213         let mut r_col_it = right.axis_iter_mut(ndarray::Axis(0));
214         if pos != 0 {
215             ndarray::Zip::from(l_col_it.nth(0).unwrap())
216                 .and(l_col_it.nth(pos - 1).unwrap())
217                 .for_each(std::mem::swap);
218             ndarray::Zip::from(r_col_it.nth(0).unwrap())
219                 .and(r_col_it.nth(pos - 1).unwrap())
220                 .for_each(std::mem::swap);
221             l_col_it = left.axis_iter_mut(ndarray::Axis(0));
222             r_col_it = right.axis_iter_mut(ndarray::Axis(0));
223         }
224         let l_piv = l_col_it.next().unwrap();
225         let r_piv = r_col_it.next().unwrap();
226         for (mut l, mut r) in l_col_it.zip(r_col_it) {
227             if l[0] == 1 {
228                 l -= &l_piv;
229                 l.mapv_inplace(|x| i8::abs(x % 2));
230                 r -= &r_piv;
231                 r.mapv_inplace(|x| i8::abs(x % 2));
232             }
233         }
234 
235         let (_, left) = left.split_at(ndarray::Axis(0), 1);
236         let (_, right) = right.split_at(ndarray::Axis(0), 1);
237         let (_, left) = left.split_at(ndarray::Axis(1), 1);
238         recursive_gauss(left, right)
239     } else {
240         let (_, left) = left.split_at(ndarray::Axis(1), 1);
241         recursive_gauss(left, right)
242     }
243 }
```

References

[1] Wikipedia. Prime number theorem. https://en.wikipedia.org/wiki/Prime_number_theorem, 2023. [Online; accessed 2-November-2023].

[2] Wikipedia. Quadratic sieve. https://en.wikipedia.org/wiki/Quadratic_sieve, 2023. [Online; accessed 24-November-2023].

[3] Wikipedia. Sieve of eratosthenes. https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes, 2023. [Online; accessed 24-November-2023].